ECE 344L Laboratory 2

Nested Routines

SPRING 2020

AUTHOR: DAVID KIRBY

DUE DATE: 25 FEBRUARY 2020

Laboratory 2
MIPS Software Development

Due Date: 25 February 2020 Name: David Kirby

Points: 100 Points
Work individually.

Objective: The purpose of this laboratory is to expand your MIPS programing skills by learning to implement nested routines. You will be using the stack to pass operands to, and return results from, a routine. You will also use the stack to save return address information, so that nested routines can be implemented properly. You will generate MIPS assembly language source code that will be assembled, downloaded, and executed. The MIPS instruction set reference sheet that was distributed in class will be very useful for this software development process. (All class slides and reference materials are posted on UNM Learn.)

Activities: For this assignment, you will write a **main** routine in which two n x 1 vectors are defined in a similar manner to how numbers_to_use: is implemented in lab 0***. (n must be equal to 6 and some of the values must be negative numbers – no values should be 0).*** Your main program will configure the stack pointer appropriately and pass the parameters to a function called **dot_product**. ***Do NOT use registers to pass the parameters to the dot product function and do NOT pass pointers to the arrays. References to the input arguments and output values will all be relative to the stack pointer. You will use the stack to return the dot product result to main, as well as the two vector average values.***

You will generate a function named **dot_product** which takes elements from each vector and calculates the dot product. (Good programming practice will include overflow checks for the multiplication, if you choose to use large value for the vector elements.) The dot_product function will call a function named **average,** which calculates the average value of the elements in each individual vector. Your dot-product routine will need to preserve the return address register on the stack before calling the average function and restore the return address after the average function executes.

You may use the $vx registers to return the value(s) from the function average value.

Upon completion of the dot-product calculations, your main program should have a "spin" loop at the end so that you can use the debugger to verify that your program has correctly calculated the vector dot product.

For this lab, you will be graded on how well you document your code. You will be well served to spend a little time generating pseudocode that shows the high level detail of what your code is supposed to be doing. Spend some time generating the design before you actually start writing code. Include your pseudocode in your lab report.

Your code will be examined to verify that you strictly adhere to the MIPS register usage conventions. **It is mandatory that you use the stack to pass parameters to the called routines as specified above, as this is the one of the main objectives of this lab.**

Introduction

The objective of this assignment was to learn how stack operations are done and to give us more practice with MIPS assembly language by learning to implement nested routines. In C, this is equivalent to nested for loops. The challenge for this lab, however, was to use stack pointers instead of storing information directly in registers. This lab was an introduction to push, pop, and register addressing, as well as more practice with memory addressing.

Solution Methodology

For this assignment, we wrote a main routine in which two 6×1 vectors were defined in a similar manner to how `numbers_to_use`: was implemented in lab 0, with some of the values being negative numbers. We configured the stack pointer and passed the parameters to a function called `dot_product` which returned the result to main, as well as the two vector average values. All of this was done with overflow checking in mind. (see Source Code). With the routine built, we attached and programmed the chipKit Pro MX4 board and ran the instructions. Initially, the program performed the mathematical functions and went into an infinite loop (see Figure 1). Below is an example of the pseudocode used to design this recursive operation.

```
int dot (int i)
{
     if (i < 6)
          vector1(i)*vector2(i);
     else sum(vector1(i)+vector2(i));
}
```

One of the reasons for using the stack is to avoid delay hazards which results in registers being called before they are finished being executed. One solution is to push all registers that must be preserved onto the stack, just as we did with the saved registers in lab 0. The caller pushes any argument registers (`$a0-$a3`) or temporary registers (`$t0-$t9`) that are needed after the call. The callee pushes the return address register `$ra` and any saved registers (`$s0-$s7`) used by the callee. The stack pointer `$sp` is adjusted to account for number of registers placed on the stack. Upon the return, the registers are restored from memory and the stack pointer is readjusted.

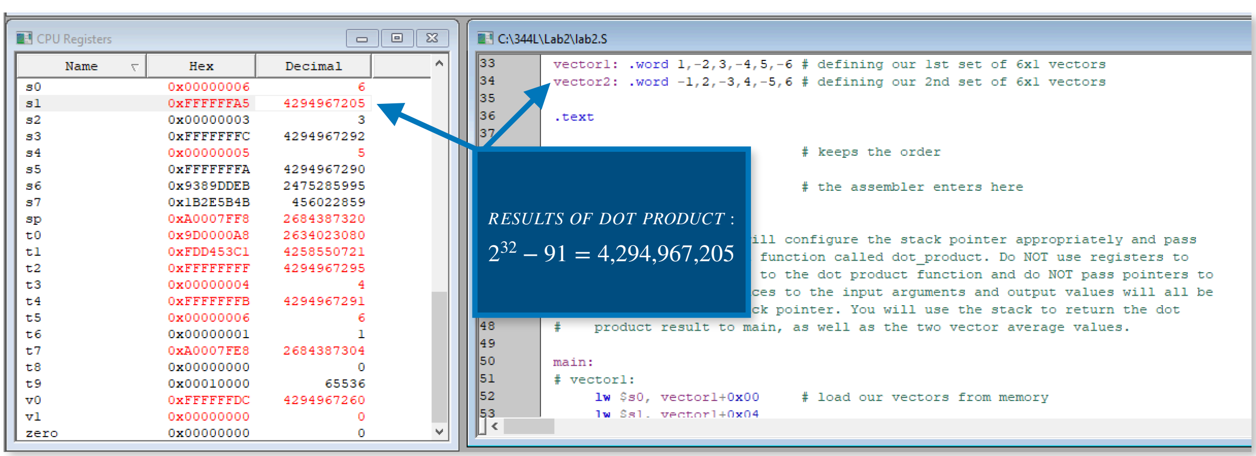

FIGURE 1

These values are consistent with the calculations as performed in MATLAB (Figure 2).

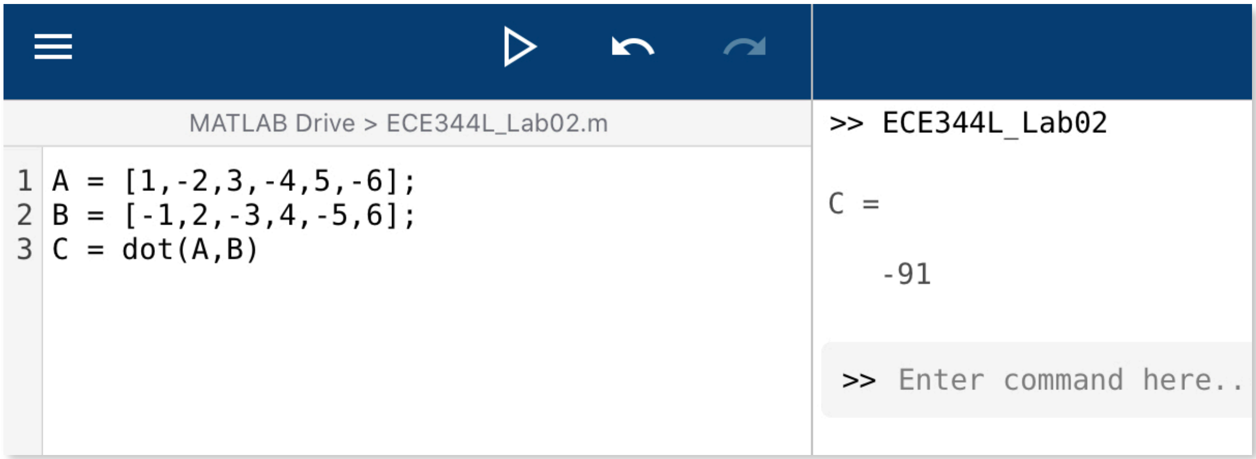

FIGURE 2

Source Code

```
/*****************************************************************************/
/*                                                                           */
/*          ECE 344L        -    Microprocessors -        Spring 2020        */
/*                                                                           */
/*          lab2.S          MIPS Assembly Nested Routines                    */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*          Author:     David Kirby                                          */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*          Detailed File Description:                                       */
/*              Implements stack operations to perform dot products on       */
/*              vectors.                                                     */
/*                                                                           */
/*****************************************************************************/
/*                                                                           */
/*          Revision History:                                                */
/*              Adjusted overflow detection                                  */
/*                                                                           */
/*****************************************************************************/

/*#include <p32xxxx.h>*/    # only include this C header if NOT using $ notation
.globl main

# Part 1:
#   Write a main routine in which two n x 1 vectors are defined in a similar
#   manner to how numbers_to_use: is implemented in lab 0. (n must be equal to
#   6 and some of the values must be negative numbers - no values should be 0).

.data
vector1: .word 1,-2,3,-4,5,-6     # defining our 1st set of 6x1 vectors
vector2: .word -1,2,-3,4,-5,6     # defining our 2nd set of 6x1 vectors
```

```
.text

.set noreorder              # keeps the order

.ent main                   # the assembler enters here

# Part 2:
#   Your main program will configure the stack pointer appropriately and pass
#   the parameters to a function called dot_product. Do NOT use registers to
#   pass the parameters to the dot product function and do NOT pass pointers to
#   the arrays. References to the input arguments and output values will all be
#   relative to the stack pointer. You will use the stack to return the dot
#   product result to main, as well as the two vector average values.

main:
# vector1:
    lw $s0, vector1+0x00    # load our vectors from memory
    lw $s1, vector1+0x04
    lw $s2, vector1+0x08
    lw $s3, vector1+0x0C
    lw $s4, vector1+0x10
    lw $s5, vector1+0x14
# vector2:
    lw $t0, vector2+0x00
    lw $t1, vector2+0x04
    lw $t2, vector2+0x08
    lw $t3, vector2+0x0C
    lw $t4, vector2+0x10
    lw $t5, vector2+0x14

# stack pointer configuration:
    addiu $sp, $sp, -64      # allocate room on the stack (12 digits + 4 temp)
    # vector1:
        sw $s0,  4($sp)     # push our vectors to the stack
        sw $s1, 12($sp)
        sw $s2, 20($sp)
        sw $s3, 28($sp)
        sw $s4, 36($sp)
        sw $s5, 44($sp)
    # vector2:
        sw $t0,  8($sp)
        sw $t1, 16($sp)
        sw $t2, 24($sp)
        sw $t3, 32($sp)
        sw $t4, 40($sp)
        sw $t5, 48($sp)

    addi $t7, $sp, 0        # record our current stack position
    jal dot_product         # jump and link our return address
    nop
    addiu $sp, $sp, 64       # after the jump, push result to stack, & end
    b end_m
    nop

dot_product:
    add $s0, $zero, $0
    add $s1, $zero, $0
```

```
    b average
    nop

average:
    lw $t0, 4($t7)          # pops vector1 from stack
    lw $t1, 8($t7)          # pops vector2 from stack

    beq $s0, 6, end_math    # end if we've reached our 6th value
    nop
    sw $ra, 52($sp)         # else store return address

    jal multiply            # then jump to the multiplication part
    nop
    lw $v0, 60($sp)         # after jump, pops return value from stack
    bltz $v1, end_m
    nop

    addi $a0, $s1, 0
    addu $s1, $v0, $s1      # accumulate multiplications into $s1

    jal overflow
    nop

    lw $ra, 52($sp)
    addi $s0, $s0, 1
    addi $t7, $t7, 8        # check the average status
    b average
    nop

multiply:
    mult $t0, $t1           # LO = (($t0 * $t1)<<32) >> 32;HI = ($t0 * $t1)>>32;
    mflo $t0                # $t0 = LO
    mfhi $t1                # $t1 = HI (store HI into a register to be checked)
    sra $t2, $t0, 31        # shift right arithmetic by 31

    bne $t2, $t1, of_else   # checks HI = 0, branch to overflow else
    nop

    sw $t0, 60($sp)         # if no overflow, push to stack

    li $v1, 0

    jr $ra
    nop

of_else:
    li $v1, -1              # if overflow, set $v1 to -1
    li $v0 , 0              # if overflow, set $v0 to 0
    b end_m
    nop

end_math:
    jr $ra                  # after multiplication, return address
    nop

overflow:
    bgtz $v0, positive
    nop
```

```
    b negative
    nop

positive:
    bltz $a0, of_good       # overflow good
    nop
    bltz $s1, of_else       # overflow else
    nop

    jr $ra
    nop

negative:
    bgtz $a0, of_good       # overflow good
    nop
    bgtz $s1, of_else       # overflow else
    nop

    jr $ra
    nop

of_good:                    # if overflow good, jump back to $ra
    jr $ra
    nop

end_m:
    nop                     # creates an infinite loop
    b end_m
    nop

.end main
```

Conclusion

Laboratory 2 was designed to familiarize students with stack operations and to give us more practice with MIPS assembly language by learning to implement nested routines. This was critical to understanding how to properly use push and pop to store and retrieve data as necessary. We also again made use of various registers and the HI and LO portions of the multiplication function to detect overflows.